Logging Kernel Events

Tera Computer Company
2815 Eastlake Avenue East
Seattle, WA 98102

December 4, 1995

1 Introduction

This document describes a feature to enhance the debugging facilities of the Tera operating system. In particular, it details a mechanism to log kernel events with minimal overhead and in a way that is non-intrusive to the developer or system administrator.

Tera kernel tracing currently consists of a large set of macros that under certain conditions, result in a message being displayed on the console. The conditions which cause a kernel trace message to be displayed involve the compile-time define `TOS_DEBUG` as well as a set of flags that are checked at runtime. These runtime flags are contained in the `DebugFlags[]` array and can be used to enable or disable three categories of traces; terse, verbose, and prolog. A developer would enable the terse traces of a given system component to get the overall gist of processing in that area. If more detailed information is needed, the verbose flag is enabled. Finally, the prolog flag can be used to trace C++ member function entries and exits (another compile-time define, `TOS_DBG_PROLOG`, comes into play here - see `debug.w` for more details). In addition, if further control is desired, the `OS_CCB_k::traceFlag` can be set to enable tracing on a per-chore basis.

The use of `printf()`s to debug code is only useful if the amount of information being relayed is minimal. Unfortunately, trace output from TOS (or any operating system for that matter) is not minimal. While it may suffice while debugging the operating system within a simulator, it should be apparent that the use of `printf()`s on the real hardware will be more difficult if not impossible. In addition, sites will not want these traces flooding the console screen but there must still be some way to gather trace information when a site sends in a dump for analysis.

For these and other reasons, a kernel event logging mechanism will be implemented along with the means of controlling it and displaying its output in a comprehensive way. Since events may be logged at a very low-level of the kernel where performance is crucial and not all features of TOS may be available, the logging mechanism will be kept simple and consist of a set of circular buffers to store pertinent information.

The following sections will give the requirements of this feature as well as detail the actual implementation.

2 General Requirements

The original requirements and background to OS event logging can be found in `~os/design/eventTracing.eve`
While more details can be found in that document, the following is a summary of the requirements:

- Minimal overhead.
- Dynamic enable/disable of logging events.
- Static configuration of number of buffers and their sizes.
- Customizable buffer management policies.
- Live system and dump file event processing capabilities.

3 User Level Event Tracing

The Performance and Debugging group currently has an event logging mechanism for the user level runtime as well as user applications. The original intent of this feature was to leverage off of this work and reduce duplicated effort.

The runtime Event Tracing implementation (see `~system/doc/perf/perf.ps`) provides the ability to register filter functions for specific logging identifiers as well as an optional guaranteed logging of all events to a specified file. Since the main requirement of this feature is minimal overhead, logging all kernel events to a file (at least from within the kernel) is not practical. Not only would this be an expensive operation, but there would be certain places in TOS where file system access would be unavailable (at boot time for example). While this is an optional feature of Event Tracing, the registration of filter functions which would be invoked upon every trace request is also unnecessary overhead. In fact, only a small fraction of the user level event logging code is needed for the OS (approximately 10 lines of buffer management code).

For these reasons, a new set of operating system functions and macros will be created to provide OS event logging. It will use some of the low level event logging code from the user level feature, but will also contain kernel specific optimizations and display routines. The record header format of the log records will be compatible with the user level event feature in order to make use of any visualization tools that are developed for it.

4 Event Buffer

Since the event logging feature must be usable by both C and C++ code, C structures and functions will be implemented.

The primary structure that controls the event buffer is `KLog`. The format of this structure is as

follows:

```
typedef struct _KLog {
        int64           subsystem;
        char*           name;
        int64           num_recs;
        int64           rec_size; /* # words per record */
        int64           header; /* Formatted header for EvtTr compat */
        int64*          buffer;
        int64           buffer_size_1; /* Total number of words in buffer - 1*/
        int64           next_write;
} KLog;
```

Each KLog buffer will contain events from a specific subsystem. The numerical identification of the subsystem is saved in the `subsystem` field of the `KLog` structure and will be one of the existing debug classes (i.e. `DBG_SCHED`, `DBG_FS`, etc.). The `name` field contains the string representation of the subsystem which will be the subsystem name without the `DBG_` prefix. For example, the name of the `DBG_SCHED` subsystem will be "SCHED".

The `rec_size` field contains the size in `int64` words of each event record. This is currently hard-coded to 8 words. The `num_recs` field contains the number of events that this particular subsystem can hold in its circular buffer. There is a set of defines that determine the size of each subsystem's buffer. These defines will need to be tuned as experience with kernel logging progresses.

The `header` field is written as the first word of each event record. It is there for compatibility with the user-level event logging feature and contains a magic number, size of the record, and record identifier which in this case is the subsystem number.

As expected, the `buffer` field points to the top of this subsystem's circular buffer. The buffer itself is currently statically allocated in supervisor memory. Additional work will be needed to support dynamic reconfiguration of buffer sizes. The `buffer_size_1` field contains the total size of the buffer in words minus one. The current implementation requires power-of-2 size buffers and thus `buffer_size_1` is used to wrap the current record pointer using a single AND operation. This restriction could be removed by using the modulus operation if performance is not impacted.

Finally, the `next_write` field is used as the current pointer into the buffer. This field is atomically incremented using the `TERA_INT_FETCH_ADD()` operation to insure buffer integrity.

The global array of `KLogs`, one per subsystem, will be statically allocated and initialized so that event logging can take place immediately upon kernel startup. To make things easier to initialize, a macro will be available so that the declaration will look like so:

```
KLog KLogs[DBG_MAX] = {
        KLOG_CREATE(NONE),
        KLOG_CREATE(PARCHORE),
        KLOG_CREATE(FS),
        KLOG_CREATE(SCHED),
        KLOG_CREATE(VM),
        ...
```

The KLOG_CREATE macro will expand to fill in all the correct data into each KLog field.

The buffers themselves are allocated from one large buffer named KLogBuffer. This buffer is also located in supervisor memory and is declared as follows:

```
int64 KLogBuffer[KLOG_REC_SIZE * KLOG_TOTAL_RECS];
```

Where KLOG_TOTAL_RECS is the sum of all subsystem records. The only piece of data that could not be statically initialized in each KLog structure is the buffer pointer. Therefore, all KLogs are statically initialized to point to the start of the KLogBuffer. Then, at runtime, the klogInit() function invoked from debug_init() sub-divides the KLogBuffer and updates each KLog subsystem's buffer pointer. Any kernel event tracing that occurs before klogInit() is not lost though as it will be placed in the DBG_NONE's subsystem buffer.

5 Event Records

An event record is generated by using one of the KLOG macros (explained in a later section). The record is stored in a circular buffer referenced from the KLog structure. Each record contains 8 words of data as follows:

```
typedef struct _KLogRec {
        uint64          header;         /* MAGIC:RECSIZE:subsystem */
        uint64          id;             /* thisChore() */
        uint64          proc_dom;       /* thisProcessor():thisDomain() */
        int64           clock;          /* teraClock() */
        int64           desc;
        int64           data1;
        int64           data2;
        int64           data3;
} KLogRec;
```

The first 4 words of the record are event independent. The header word is filled in with the header field of the corresponding KLog structure. It consists of a magic number, record size in words, and the subsystem identifier. It is stored for compatibility with the user level event tracing feature.

The next word contains the chore control block pointer of the chore logging the event (thisChore()). The proc_dom word contains the processor number and domain that the chore is running in. The combination of these two words identifies a unique chore. The clock field contains a time stamp of the event as returned by teraClock(). This allows debugging utilities to properly merge all subsystem logs into the correct chronological sequence.

The last four words of the record are event specific as specified by the parameters to KLOG(). The first, desc, is the character pointer to the format string describing the event. The data1, data2, and data3 words contain the optional pieces of data associated with the event.

6 Enabling/Disabling OS Event Logging

Kernel tracing is currently accomplished through debugging `printf_k` statements sent to the system console. Whether a debugging statement is printed or not depends on two things; the intensity level specified by the trace statement and the settings of the `DebugFlags` array.

The existing kernel `printf` traces are controlled through fields in the `DebugFlags` array, indexed by subsystem id. This feature adds two new flags, `d_klog` and `d_klogprintf` to the `Dbg_Intensity` typedef:

```
typedef struct _Dbg_Intensity {      /* debugging intensity */
    unsigned int   d_prolog:1;       /* log function entry/exit */
    unsigned int   d_terse:1;        /* important trace messages */
    unsigned int   d_verbose:1;      /* detailed trace messages */
    unsigned int   d_klog:1;         /* enable event logging */
    unsigned int   d_klogprintf:1;   /* print events to the console */
} Dbg_Intensity;
```

If the `d_klog` flag is true, `KLOG()` events from this subsystem will be logged in the appropriate buffer. By default all subsystems will have the `d_klog` flag enabled. If `d_klogprintf` is true, the `KLOG()` events from this subsystem will be printed on the console if the appropriate intensity flag is also set. This way, the existing kernel traces can be replaced by `KLOG()` requests without losing the ability to see the trace on the console (see examples in the following sections).

In addition to the `d_klog` flag check, this feature also includes a log level check. A new global variable, `klogLevel`, controls which events get recorded. The current log levels are the following:

```
enum {
    KLOG_OFF,          /* 0: No KLOG event logging */
    KLOG_TERSE,        /* 1: Terse events logged */
    KLOG_VERBOSE,      /* 2: Verbose events logged */
    KLOG_PROLOG,       /* 3: Prolog events logged */
    KLOG_ALL           /* This enum should always be last */
};
```

It is important to note that the check for the `d_klog` flag and other `d_xxx` intensities is only done if `TOS_KLOG_DEBUG` is defined. If it is not defined, the intensity level given on the `KLOG()` invocation will only be checked against a static level of our choosing (i.e. `KLOG_TERSE`). This will decrease the performance impacts of the logging feature by effectively removing all `KLOG()` invocations above the chosen level.

7 Logging an Event

The actual insertion of a log record will be done through the KLOG macro set created by this feature. The format of the macro is:

```
KLOG(subsystem, level, desc, d1, d2)
```

The subsystem parameter is one of the DBG_xxxx enumerated constants. The level argument should be one of the klog intensity constants such as KLOG_TERSE. The desc is a pointer to the text format describing the event. Finally, d1 and d2 are the additional data to be logged with this event.

For example, the following two traces would produce the same result except one would be recorded for later playback:

```
DBG_SCHED_V(("postProcess: task:%d curPD:%d\n",taskIndex,curPD));

KLOG(DBG_SCHED, KLOG_VERBOSE, "postProcess: task:" HEX64FORMAT
                " curPD:" HEX64FORMAT, taskIndex, curPD)
```

Note: The reason HEX64FORMAT is used as the formating option is so that the correct data is displayed in the Awesime world. Since each data item under awesime is stored in 32 bit words, a %d conversion would only display the top 32 bits of the 64 bit data item. Using HEX64FORMAT insures the proper data is displayed under both worlds.

To make life easier (or more difficult depending on how you look at it), there are numerous KLOG macro variations to shorten the actual KLOG invocation. Available KLOG macros are:

```
KLOG[TVP][012](area, [level,] desc, [d1,] [d2])
```

The [TVP] characters represent KLOG_TERSE, KLOG_VERBOSE, and KLOG_PROLOG respectively and if used, negate the need for the level parameter. If no character is specified, then the level specification must be given.

The [012] digits indicate the number of data items to be logged. If '0' is used, d1 and d2 are not given, if '1', only d1 is specified, and if '2', both d1 and d2 must be given. If no digit is specified, '2' data items are assumed.

With these macros in mind, it should be easy to see how the existing TRACEK macro could be modified to use KLOG():

```
#define TRACEK(area,string) KLOGP0(area, string);
```

Here are some more examples of existing trace code and the corresponding KLOG() invocation that

could replace it:

```
DBG_SCHED_T(("doloadTask -> %s\n",ccb));

KLOGT1(DBG_SCHED, "doloadTask -> %s",KLOG_STR(ccb));


IF_DEBUG(DBG_VM,d_verbose) {
    printf_k("DataSegment_k::extend succeeded\n");
}

KLOGV0(DBG_VM, "DataSegment_k::extend succeeded´´);


IF_DEBUG(DBG_NET, d_verbose) {
        addLog(&NetLockTrace, "read lock ", file, line);
}

KLOGV(DBG_NET, "read lock %s, line %d´´, KLOG_STR(file), line);
```

Note the use of the `KLOG_STR()` macro above. When running the Tera system, this macro does nothing. But, when simulating under Awesime, the macro moves the character pointer to the upper 32 bits of the `int64` word. This allows `%s` formating to pick up the correct string address.

8 Trace Tools

The `klogPrint(subsys, type)` function is provided by this feature to view the contents of the trace buffers. The `subsys` parameter indicates which subsystem buffer you would like dumped. If the subsystem given is 0, all trace buffers will be merged and dumped in reverse chronological order (most recent to least recent traces).

The `type` parameter indicates how the data should be displayed. Here is an example of output from merged traces using the `KLOG_PRINT_FULL` display type:

```
[KP:0,0,50000010][0,0,ALARM] Entering function "AlarmID_k :: AlarmID_k"
[PC:0,0,300001bf][0,0,KLIB] Entering function "klib_entry"
[PC:0,0,300001bf][0,0,IPC] Entering function "Message::Message"
[PC:0,0,300001bf][0,0,UTIL] Entering function "HashChainInt::hash"
[PC:0,0,300001bf][0,0,UTIL] Entering function "HashChainInt::insert"
[KP:0,0,50000010][0,0,ALARM] Entering function "AlarmPool_k :: free"
[KP:0,0,50000010][0,0,UTIL] Entering function "HashChainInt::hash"
[KP:0,0,50000010][0,0,UTIL] Entering function "HashChainInt::remove key"
```

In addition, a user-level program (`crash(8)`?) will be developed to gather all kernel trace buffers, merge them in chronological order, and display them in human readable form. This tool may be developed by the kernel group or perhaps, since the format of these buffers is compatible with the

user level Event Tracing feature, tools developed for that product could also be used on kernel buffers.

A Issues and Alternatives

A.1 Trace Description

Logging a pointer to a string obviously assumes more than the trace buffer itself must be available for display purposes. If this proves to be a problem, here are a couple of alternatives:

- A short 8 character trace description can be logged instead of a pointer to a full character array.
- A minor event ID could be logged instead of the character pointer. This minor number would then correspond to some text string declared elsewhere.

  While this is a viable option, it may prove to be a bother maintaining the trace and description in two separate places.

A.2 Buffer Distribution

As mentioned, the intention is to have a trace buffer per major subsystem. Other possibilities exist such as grouping a number of subsystems into one trace buffer or having a trace buffer per processor.

In addition, the buffers themselves will initially be static since I'm sure there will be a number of traces occurring before dynamic kernel memory allocations are available. Obviously this will be a waste of space if dynamic resizing of the buffers is made available. This issue can be addressed at that time.

A.3 Performance Considerations

This feature will require a good deal of tuning to be sure the performance impact is minimal. Evaluations will need to be made on the logging level the system should be released at as well as the size of each subsystem's buffer.

The contents of the buffer are important considerations too. While the feature is currently logging the full 2 word `OS_ChoreID_K`, perhaps this should be changed to save just the CCB pointer. This will require only one store operation verse at least 3 loads and stores when using the `getChoreID()` function.